AD-A197 342

RADC-TR-88-7
Final Technical Report
January 1988

GRAPHICAL PROGRAMMING AND MONITORING

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
Arpa Order No. 4469

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

DTIC
ELECTE
JUL 0 8 1988
D
E

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

88

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-7 has been reviewed and is approved for publication.

APPROVED: Richard M. Evans

RICHARD M. EVANS
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

DTIC
COPY
INSPECTED
6

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

GRAPHICAL PROGRAMMING AND MONITORING

J.C.R. Licklider

Contractor: Massachusetts Institute of Technology
Contract Number: F30602-82-K-0170
Effective Date of Contract: 17 Sep 82
Contract Expiration Date: 20 Sep 86
Program Code Number: 5E20
Short Title of Work: Graphical Programming and Monitoring
Period of Work Covered: Sep 82 - Sep 86

Principal Investigator: J.C.R. Licklider
Phone: (617) 253-7705

RADC Project Engineer: Richard M. Evans
Phone: (315) 330-3564

Approved for public release; distribution unlimited

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Richard M. Evans (COES) Griffiss AFB NY 13441-5700 under Contract F30602-82-K-0170.

AIR FORCE 86145/ 10-6-88 - 70

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) LCS- 1986/AF 1	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-7
6a. NAME OF PERFORMING ORGANIZATION Massachusetts Institute of Technology 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Romc Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) Laboratory for Computer Science 77 Massachusetts Avenue Cambridge MA 02139	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Projects Agency 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-K-0170
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd Arlington VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: PE61101E; PROJECT NO: D469; TASK NO: 01; WORK UNIT ACCESSION NO: 01
11. TITLE (Include Security Classification) GRAPHICAL PROGRAMMING AND MONITORING	
12. PERSONAL AUTHOR(S) J.C.R. Licklider	
13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Sep 82 TO Sep 86
14. DATE OF REPORT (Year, Month, Day) January 1988	15. PAGE COUNT 72
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Graphical Programming graphical monitoring
Iconic programming
Diagrammatic programming

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The goal of this research was to explore graphical programming and graphical monitoring of the interpretation of computer programs and develop a concept demonstration system for creating programs graphically. The system, GRAPPLE, is essentially a graphical interface to a LISP-like language called "MDL." Using GRAPPLE, the programmer develops a diagrammatic representation of a program by making selections from menus with a "mouse" and typing some short labels on a keyboard. GRAPPLE then creates a symbolic MDL program as a subprogram in further programming or run it with whatever data it requires. The report describes the programming process in detail and includes a discussion of issues, conclusions and recommendations.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard M. Evans	22b. TELEPHONE (Include Area Code) (315) 330-3564 22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Table of Contents

GRAPHICAL PROGRAMMING AND MONITORING

A Graphical Interface to a LISP-like Programming Language

1. Foreword

This project was initiated as part of an effort, sponsored by the Defense Advanced Research Agency and managed for DARPA initially by Craig Fields and Clinton Kelly, to improve the art of computer programming and the understanding of programs through the use of graphics and visualization. During the latter part of the project, the program manager for DARPA was Stephen Squires. The project was contracted for and monitored by the Air Force under Contract No. F30602-82-K-0170. The project monitor throughout was Richard M. Evans of Rome Air Development Center. The M.I.T. participants in the project greatly appreciate the sustained and sustaining interest and contributions made by Fields, Kelly, Squires, and Evans.

Much of this project was devoted to exploration of ideas about how to represent program constructs graphically and how to use graphics to facilitate the development and to increase the understandability of programs. Some of the ideas that seemed most developable, given the present state of computer graphics, concerned the use of icons and structure diagrams in the creation of programs. Some of those ideas were incorporated into a system called *GRAPPLE*, a graphical programming language or, more accurately, a graphical interface to MDL, which is a derivative of the programming language LISP. GRAPPLE is described in the section on A Graphical Programming System. MDL is described in *The MDL Programming Language* by S. W. Galley and Greg Pfister.

The participants in the project were 18 M.I.T. undergraduate students, who worked for periods varying from a few months to a little over three years, and a professor of computer science, who functioned as principal investigator as well as experimenter and programmer and participated throughout the project. The 18 undergraduate students were Jeff Dike, Malcolm D. A. Duke, Rolf Embom, Mark A. Herdeg, Janet Hirata, Colleen Humphreys, Young-Jo Kim, Marie Paz Kudich, Bassanio W. C. Law, Stuart Malone, Mark McEntee, Catherine Naylor, Gordon S. Shaw, Colin L. Shepard, John Shrivanandon, Bosco Y. So, Michael A. Thompson, and Jennice Y. Wang. The principal investigator was

J. C. R. Licklider, the writer of this report. The allocation of tasks among the participants is outlined in the section, Who Worked on What.

2. Introduction

2.1. Purpose of Project

The purpose of the project, Graphical Programming and Monitoring, was to explore the use of computer graphics in preparing programs and in monitoring the interpretation or execution of programs. The graphic techniques employed were limited to line drawings and to small icons composed of binary dot or pixel patterns in a matrix of about 10 by 16 pixels. The graphics were supplemented where necessary by alphameric text. The programming language used throughout the project, and the language in which the graphical programming system developed by the project prepares programs, is MDL, a close relative of LISP. The acronym *MDL* can be expanded into *More Datatypes than Lisp.*

2.2. Three Main Undertakings

During the 4-year course of the project, there were three main undertakings. The first was the selection or development -- the latter, as it turned out -- of a graphics system within MDL, a system to facilitate the preparation of graphical programs. The second was the exploration of a variety of graphical ideas and techniques that offered promise of facilitating the preparation or the understanding of ordinary (nongraphical) programs. The third was the development of a concept demonstration program, a graphical programming system for preparing MDL programs, a system that carries one small set of graphical ideas and techniques far enough to permit the preparation of nontrivial *MDL* functions and macros and to let one experience some of the advantages and limitations of the graphics medium.

2.3. Who Worked on What

The people who worked on the project were eighteen M.I.T. undergraduate students and one professor. The students were associated with the project for periods ranging from one term to the duration and the professor, of course, for the duration. In the following table, the people who worked on the project are paired with the activities to which they mainly contributed.

Jeff Dike	A system to let VAX-BitGraph programs run on a VAXstation II. (Thesis)
	A system for recording graphical scope displays on the Xerox Dover laser printer.
	Programs to exploit capabilities of the BitGraph display.

Malcolm D. A. Duke
Programs to facilitate testing and documentation of other programs.

Rolf Embom — Graphical display of data structures at various levels of abstraction.

Mark A. Herdeg — Graphical support software.

Dynamic graphical display of evaluation of arithmetic and algebraic expressions.

Janet Hirata — Dynamic graphical display of interpretation of MDL functions.

Colleen Humphreys
Dynamic tracing program.

Updating early programs and preparing demonstrations.

Copying the project's computer software onto back-up tapes.

Young-Jo Kim — Exploring approaches to graphical representation of data types and operations.

Marie Paz Kudich — Exploring approaches to graphical representation of data types and operations.

Bassanio W. C. Law
Documentation standards and documentation aids for the project's software.

Stuart Malone — System wizard.

Graphical support software.

Documentation support software.

Dynamic graphical display of evaluation of arithmetic and algebraic expressions.

Mark McEntee — Exploring approaches to graphical representation of data types and operations.

Catherine Naylor — Graphical representation of MDL programs as trees. (Thesis)

Gordon S. Shaw — Exploring approaches to graphical representation of data types.

Colin L. Shepard — Exploring approaches to graphical representation of data types.

John Shrivanandon
The representation of programmer-defined data types in graphical

programming. (Thesis)

Bosco Y. So — A system for loading MDL functions and data sets when they are referred to during interpretation or execution.

Michael A. Thompson — Graphically representing the control structure of MDL programs. (Thesis)

Jennice Y. Wang — A library of MDL functions.

J. C. R. Licklider — Dynamic graphical display of the interpretation of the application of MDL functions to data.

The graphical programming system, GRAPPLE.

2.4. The programming language MDL

A decision made early in the project was to use the programming language MDL as the language in which to write the project's programs and the language in which the project's graphical programming programs would write programs. MDL is a derivative of LISP, and the decision had two parts: (1) the decision for a member of the LISP family of languages, and (2) the decision for MDL in particular. The governing consideration for the first part was to have a language in which the parts of a program are data that programs can recognize and operate upon more meaningfully than merely as strings of characters. The only candidates with that characteristic were members of the LISP family and Smalltalk, and the latter was just then emerging from Xerox Palo Alto Research Center, not mature enough for us to use effectively. The governing considerations for the second part were (1) extensibility in respect of data types and (2) familiarity. MDL was clearly superior to the other LISP-like languages then available in respect of data type extensibility; it was used extensively by other members of the group in which the project was embedded; and it was very familiar to the principal investigator.

There was indeed, however, a consideration that did not favor MDL: MDL was, at the outset of the graphical programming and monitoring project, in the middle of the process of being thoroughly rewritten in such a way as to make it easy to bring up on diverse computers. The objective was attractive, of course, but the idea of putting all one's programming at the mercy of a language under revision was not. We decided to proceed with MDL anyway because we knew the revisers well and had great confidence in their capability and their willingness to help users-in-need. As it has turned out, MDL and its revisers served us well. We have been the first to encounter a few revision-implanted bugs,

but only a few, and on the whole MDL has given us a very solid foundation.

2.5. The Computer System

In the early months, the project used a DECSYSTEM-20 time sharing system from the Digital Equipment Corporation and BitGraph terminals from BBN Computer Co. The time sharing system was not good for graphics programming, of course, but it was adequate as an interim system. The BitGraph terminals were, when they worked, good for our purposes. The most important characteristic of the display, for us, was its resolution, and the BitGraph gave us a display of 1024 by 768 binary pixels. The BitGraph supported a mouse, which is the most popular (if not the best) graphical input device, and which was quite adequate for our purposes. In addition, of course, the BitGraph provides a keyboard, and the keyboard has 21 keys that can be used as function keys.

Toward the end of the first year, the project acquired a Digital Equipment Corporation VAX-750 computer with 8 megabytes of semiconductor memory and a Fujitsu Eagle disk. The VAX was operated under the Unix operating system with the Berkeley shell (version 4.2 or 4.3). The VAX was operated as a time sharing system, but usually with only one to four users, and with the arrangement that a user in need of a lot of machine cycles usually could command 80 or 90 per cent of all the VAX had to offer. The new MDL operated well in that context, and the hardware/software system provided an excellent programming environment.

Perhaps to balance out some of the good fortune, the performance and reliability of the BitGraph displays began to deteriorate, and BBN Computer Co. ceased to support them with repairs or new parts. We did not want to get into a large order of display reprogramming, so we took advantage of the fact that other groups in the laboratory were moving away from BitGraphs to other displays. We collected deteriorating BitGraphs and wore them down to total collapse. Fortunately, a couple of them appear to be outlasting the project, and, while they are working, they are quite satisfactory. Indeed, by the standards of three or four years ago, when they were built, they are excellent.

3. A Graphics System Within MDL to Support Graphical Programming and Monitoring

The first approach of the project to graphics was to try out a system already available within MDL, a system called *DIGRAM* (*Device Independent Graphics for MDL*) that had been designed and implemented as a Master's thesis problem by Lim Po. This system had many features, including features that we did not need in order to draw the rather simple, diagramatic figures we planned to use, and the system paid in running speed for its extra functionality. Very early in the project, therefore, Stuart Malone and Mark Herdeg undertook the development of a rather simple graphics system to support the preparation of graphical programming and monitoring programs in MDL. The system they created, ICONOGRAPHER, will be described briefly in this section.

3.1. Regions

Whereas DIGRAM provided for displaying figures in movable and resizable windows and for translating, scaling, and rotating the figures and clipping them to make them fit (and keep on fitting) into the windows, ICONOGRAPHER avoided or isolated much of that processing -- especially the clipping, which requires a large amount of computation and is best accomplished by special-purpose hardware. ICONOGRAPHER uses a simple version of the window concept. A *region* is a structured object that contains the following information:

- the identifying number of the region
- the origin of the region in terms of whole-screen coordinates
- the height and width of the region
- the margins of the region (for displaying text)
- the display mode of the region (one of 16 Boolean functions of two binary variables, the *source* pixel and the *destination* pixel, in a scheme widely used in computer graphics but not crucial to our use of the system)
- previous display modes of the region (in a stack below the present mode)

ICONOGRAPHER provides functions for creating and destroying regions and for setting and resetting origins, heights, widths, margins, and modes. Each drawing function of ICONOGRAPHER takes a region as an argument and plots in the region's coordinate system. Thus regions provide a limited version of translation, but not of scaling or rotation, and not of clipping. If the user tries to draw a line that extends outside of a region, the line is not plotted at all.

3.2. Drawing Functions

ICONOGRAPHER provides functions for drawing points, straight lines, splines (curved lines), polygons, ellipses (filled and unfilled), and text (in the current font). And it provides functions for setting, pushing, and popping drawing modes, clearing regions, and performing so-called *mix-box* operations that depend upon what is being displayed in a source area (alluded to earlier) as well as upon what is being displayed in the destination area (in which the present plotting is to be done).

The drawing function that incorporates translation, scaling, and something approximating rotation is the function called DRAW-ICON. If given as arguments a REGION and a data type called ICON consisting of figures mentioned in the previous paragraph, it draws the figures in the specified region, using the coordinates in terms of which the figures are specified as region coordinates. If given an extra argument, a POINT consisting of two integers, it translates the icon figure or figures, using that point as their origin. If given a second extra argument, a DELTA consisting of two integers, DRAW-ICON scales the figure or figures with scale factors derived from the integers. And, if given a third extra argument, another DELTA of two integers, DRAW-ICON tilts the axes at angles derived from the integers. Thus DRAW-ICON makes it easy to define a diagramatic figure shape and then display it in various locations at various sizes and to control its orientation or distort it in the process of displaying it.

3.3. Fonts

ICONOGRAPHER includes a font editor and routines for displaying strings of characters of selected fonts at specified locations in regions. The font editor permits the user to specify the dimensions of each character in pixels and then to switch each pixel *on* or *off* with the mouse. It is possible, therefore, to create true icons in two different ways, (1) as line drawings with functions from the DRAW package and (2) as quaisi-pictorial characters with FONTED.

The computer system operates with two concurrently available fonts, either of which can be the active or default font, and any number of stored fonts. ICONOGRAPHER makes if convenient to bind or rebind any one of the stored fonts to either of two concurrently available font names. Thus one can have a fairly large set of icons on tap for immediate display and a very large set for display after a little switching that causes imperceptible delay.

3.4. The Mouse

The mice used with the computer system are three-button mice. A user -- programmer or monitor -- has one physical mouse. It is associated with the program construct called *the current mouse*. Inside MDL, in the MOUSE package of ICONOGRAPHER, there are as many software mice as needed, but only one of them can be the current mouse. Each of the software mice is associated with a region. Each region has a mouse stack onto which mice of the region can be pushed and from which they can be popped.

Each mouse has a set of parameters that govern its sensitivity to movement, the circumstances under which the mouse will report to the computer, and what will happen -- i.e., what function will be activated -- when the mouse reports. The MOUSE package provides functions for installing a mouse as the current mouse, for pushing and popping mice, and for setting or resetting the parameters.

In graphical programming, the mouse is used in two basically different ways.

The first way gives the user full initiative. The program the user is going to interact with is not running. The MDL interpreter is just sitting in a loop waiting for something to be fed to it from the keyboard or the mouse. (And we are focusing, now, on the case in which the input is going to come from the mouse.) If, for example, the mouse (together with the mouse circuitry and the mouse software in the terminal) reports that its middle button has been pushed and that its coordinates are 123 and 456, a function associated with the middle button begins to run, and it has the coordinates available to it as data. If the coordinates are associated with a particular selection in a menu, the function can examine the selection and activate another function associated with that item in the menu. Thus the user can control -- actually indirectly but seemingly directly -- what happens. We refer to this kind of control as *user's initiative* control or, if the user is programming, as *programmer's initiative* control.

In the second way of using the mouse, a program that controls in a general way what is going to happen is already being interpreted or executed by the MDL interpreter. That program instructs the user to make a selection with the mouse and then waits for the user (and mouse) to respond. The user usually responds by moving the mouse to a pertinent location and then pressing a mouse button. The program then usually carries out the general action it was set to carry out, but the action is further defined by the coordinates (or other information) read from the mouse. We refer to this kind of control as *user's responsive* or *programmer's responsive* control.

3.5. Use of ICONOGRAPHER

ICONOGRAPHER was used by all the members of the project who wrote graphics programs, and it provided the graphical substructure of GRAPPLE, the graphical programming program to be described in the second-following section. ICONOGRAPHER was judged easy to learn, easy to use, and sufficiently powerful for the needs of the project. Perhaps the stack mechanism associated with the mouse was even more than any of us needed, and surely it would have been good to have a menu package closely integrated with the mouse functions, but, on the whole, ICONOGRAPHER turned out to be just about what we needed. Unfortunately, however, it does not appear that there is any good path for ICONOGRAPHER to follow to find utility beyond the present project. It is limited to the context of MDL and the BigGraph display, and there does not seem to be for either a future full of widespread use.

4. Explorations in Graphical Programming and in Graphical Facilitation of the Understanding of Programs

This section is an effort to summarize a number of informal explorations, carried out mainly by students. Three of the explorations were carried far enough to warrant describing them individually. The others, together with informal discussions surrounding them, led to some subjective conclusions that may be worth recording.

The basic idea underlying most of the explorations in monitoring the interpretation of programs is the idea of dynamic graphical representation of programs, data, and the operation of programs on data. There are many approaches to and variations of the idea: dynamic flowcharts; trace programs that correlate flow through the text of the program with changes in the data structures on which the program operates; iconic operators acting upon iconic data; focusing attention upon the data and depicting changes in them graphically instead of textually; representing both programs and data schematically and showing explicit links between the execution of each part of the program and the data changes that are effected by the execution. Several of these approaches and variations upon them were explored during the project.

4.1. Tracing Programs

An action common to several of the approaches is tracing the program. Essentially, what one wants is to be able to examine the data, then interpret one expression of the program, and then examine the data again. Since the expressions of a program in a LISP-like language are usually tree-structured, it is good to adopt a tree-walking procedure for interpreting the program step-by-step.

4.2. A Dynamic Graphic Evaluator of Arithmetic Expressions

At the outset of the project, Stuart Malone and Mark Herdeg wrote a program that provided a dynamic graphic presentation of the evaluation of simple, tree-structured expressions. Given the expression <COS <+ <- .X .Y> <- .X .Z>>>, for example, their program showed two boxes, one for COS and the other for the rest of the expression. Then the latter box was subdivided to provide a box for the + and, to its right, two boxes for the <- .X .Y> and the <- .X .Z>. And then those two boxes were subdivided to provide boxes for the the respective operators (both instances of -) and operands (.X and .Y in the first case and .X and .Z in the second). Then the current values -- say 5, 4, and 3, respectively -- were substituted for the .Xs, the .Y, and the .Z. Then the lowest-level boxes were removed and the values -- 1 for <- 5 4> and 2 for <- 5 3> were substituted. And then the next-level-up boxes were removed and 3 was substituted for <+ 1 2>. Finally, the next-

level-up boxes were removed and -0.9899931 was substituted for <COS 3>.

The Malone-Herdeg program was simple but dramatic. It illustrated the hierarchical evaluation scheme of LISP-like languages and made it clear and obvious that evaluation proceeds first from left to right, then turns around and flows back in a leftward direction, winding up in the center of the diagram. It is easy, then, to explain what *top down* and *bottom up* mean, and to see the relation of the box diagrams to tree structures. The program had a definite limitation, however: It did not deal with the special functions of MDL, such as COND, that affect flow of control.

4.3. INFORMER

Also early in the project, Mike Thompson programmed a general tree-walking tracing program called *INFORMER*. It is described in his undergraduate thesis, Graphically Representing the Control Structure of MDL Programs. INFORMER used as its basis a scheme of type substitution that made it possible to employ the MDL interpreter without changing it at all, yet to have it evaluate supplementary expressions just before and just after evaluating selected expressions in the program. Essentially, the user selected the expressions that were to be preceded and followed by supplementary evaluations and preprocessed those expressions by changing their types to corresponding new types. These new types were endowed with rules for evaluation or application that sandwiched the old rules (that is, the rules for the corresponding old types) in between rules for the supplementary processing. Thus what would ordinarily be the invisible processing of the expression <SET X <+ .X .Y>> could be changed into a graphical representation of .X and .Y, then the invisible processing of the expression <SET X <+ .X .Y>>, and then a graphical representation of the new value of .X and the new (same as old) value of .Y, perhaps even with arrows to make it clear how two values -- the values of .X and .Y -- flowed into the sum and then one value -- the result value -- flowed from the sum back to the X to become the new .X, leaving the value of .Y unchanged.

INFORMER provided a graphical control system with which the user could isolate certain regions of the tree for concentrated examination of data changes. It provided for both textual and graphical representation of the operation of the program and its effect upon data. Thus INFORMER provided a basic tracing program upon which one could build graphical displays to test out various approaches to dynamic graphical representation. Mike Thompson exploited this capability to a limited extent, building simulators for COND, PROG, and a few other special operators and iconic representations for a few data types. His essential contribution, however, was to provide a working implementation of the dynamic

tracing paradigm that clearly demonstrated the concept and made evident some of its main capabilities and limitations.

4.4. Graphical Representation of MDL Programs as Trees

During the second year of the program, several students explored aspects of dynamic graphical representation. The project that developed the general idea farthest was one carried out by Catherine Naylor and described in her undergraduate thesis, *Graphical Representation of MDL Programs as Trees*. The action of her program was directly analogous to that of the Malone-Herdeg program, but her program used a downward-branching tree representation instead of a box representation, and it was more general, dealing with COND, AND, OR, and other special functions of MDL. The main limitation of her program was that it did not deal adequately with evaluations of large expressions that tend to "run off the screen".

4.5. A Basic Non-Graphical Tracing Function for MDL

Colleen Humphreys also developed a tracing program. Her program proved to be so useful without graphics that she worked on it mainly as a powerful and efficient textual tracer, and it became the standard tracing function of MDL.

4.6. Dynamic Icons

Several efforts were made to explore the concept of dynamic or kinematic icons. None of the efforts was able to carry the idea any large fraction of the distance it would have to be carried to create a practical system because, if the analysis were made at a low level, too much detail was created, and if the analysis were made at a high level, there were too many different actions to deal with, and the number of individual functions required to implement a system was prohibitive. In order to develop the idea just a bit, however, let us describe two of the notions explored.

4.7. A Dynamic Icon for Multiplication

On the one hand, consider the action involved in multiplying one number by another. A program was written that represented the action graphically by starting with two arrows whose lengths represent the magnitudes of the numbers. For the sake of simplicity, both numbers were assumed to be positive and oriented with arrows pointing upward. The program rotated one of the arrows until it was horizontal and set its base at the point occupied by the base of the other arrow, thus forming the x and y axes of a graph. Then the program drew two more lines to form a rectangle of height equal to the length of one

arrow and width equal to the length of the other arrow. Then the program filled up the rectangular area with a continuous line looping back and forth like a length of string to approximate filling up the rectangle. And, finally, the program made a small hole in one side of the rectangle, near the base, and pulled the string out, stretching it to make a straight line of length proportional to the area of the rectangle and therefore to the product of the two numbers.

The program thus presented a dynamic or kinematic representation of multiplying two numbers. It is conceivable that one might develop a system with such a dynamic icon for each basic operator of a programming language. But the problems of programming and monitoring programs probably do not extend down to the level of seeing the relation of the product of two numbers to the area of a rectangle. The program might be of some interest in the teaching of arithmetic, but it portrayed events at too low a level to serve as a building block for a program to facilitate the monitoring of programs during interpretation or execution. In short, at the low level of analysis chosen, there are not too many *different* operators, there are just too many operators: it takes very many subordinate operations to accomplish anything the user identifies as an objective, and the apparent complexity is overwhelming if each of them is shown in great detail.

4.8. A Dynamic Icon for Dividing a Rectangle into Rectangular Halves

On the other hand, consider the operation of dividing a rectangle, lined up with the horizontal and vertical axes, into two equal rectangles by drawing a line either horizontally or vertically through its middle. That is a somewhat (but not much) higher-level operation than multiplication. It is an operation that we happened to encounter several times in graphical programming, but if it were viewed as a member of a comprehensive library of program components, a rectangle halving function would have to be one of perhaps several thousand useful (but not all very frequently useful) functions.

Dynamic portrayal of the operation is simple and straightforward: A program was written in which the original rectangle was defined by four points, its upper left-hand corner, its upper right-hand corner, its lower right-hand corner, and its lower left-hand corner. The program was given an argument indicating whether the cut was to be along the horizontal midline or the vertical midline. If horizontal, the program determined the midpoints of the vertical lines. It displayed a bright spot at each end of the left-hand vertical line and then linked the bright spots with a double spline curve resembling a curly bracket, and another bright spot was drawn at the midpoint of the line, which was pointed to by the point of the bracket. Then the same process was portrayed at the other vertical line. And then the

horizontal dividing line was drawn, connecting the bright spots designating the midpoints. If the dividing line were to be vertical, of course, the program carried out corresponding operations on the top and bottom lines of the original rectangle.

The significant thing about this dynamic icon for subdividing rectangles, of course, is not any dramatic quality of presentation or appeal to any spatial mechanisms of understanding but the fact that, at this level of analysis, there are thousands and thousands of actions that are carried out by computer programs. It would be feasible to construct special-purpose systems -- for example to show fluids flowing through pipes and vessels -- but it would not be feasible to construct a general-purpose system to facilitate the understanding of randomly chosen programs because there would be too many different icons corresponding to the too many different operations.

4.9. Conclusions About Dynamic Graphical Representation of the Evaluation of Programs

1. The main problem, toward the solution of which, if there is one, we did not make any progress, is the problem just illustrated: the problem of finding a level of analysis high enough to focus on practically significant difficulties of understanding yet low enough not to require thousands of different icons.

2. During the project, we became aware of and interested in a graphical interaction system being developed by Don Hatfield of IBM. Hatfield's system is based on only five or six fundamental operations, which he represents graphically. Any action can be captured as an object and then operated upon by the fundamental operations. It may be that Hatfield has the right approach to the problem of complexity/diversity. At any rate it is an interesting and promising one.

3. In order to deal in a practical way with the complexity/diversity problem in the context of graphics, it is necessary to find a way of composing graphic representations of higher-level programs and data sets out of graphic representations of lower-level programs and data sets.

4. It is necessary, also, to deal both with flow of control and flow of data. The key to analysis of flow of control is program tracing.

5. Program tracing is so much easier and more powerful in an interpretive context than in the context of execution of compiled code that, surely, a graphical monitoring system should take advantage of an interpreter. Granted, interpretation ordinarily is slower than execution of compiled code. What one actually wants is what is provided by LISP and MDL, an interpreter that will execute compiled code when it comes to compiled code and that will interpret source language otherwise. Then the user can control what parts of the program are traced, what parts are sped through, by loading the compiled code of the perfected sections of the program and source language versions of the sections that need to be debugged.

6. A tracing program has to be able to deal with the special functions (such as

COND and PROG), but it also has to be simple for the user to use. It has to provide simple hooks to which the user can attach graphical and/or textual displays, and the type-changing scheme, if used, must be transparent to the user.

7. There appears to be no truly satisfactory solution to the problem of "running off the screen". One of the great promises of graphics is to keep active the perspective of the over-all program, and zooming in on details tends to lose the over-all perspective. On the other hand, even a modest program is too large to display on any available screen in full detail. The most attractive prospect is one or more large, high-resolution displays, with a display area for the over-all program that remains in view all the time, with another display area for the over-all data situation that remains in view all the time, and with other display areas that can be assigned to subordinate parts of the program and subordinate sets of data as the monitoring process proceeds.

8. Tree structures are difficult to display efficiently on conventional rectangular displays. The usual tree-shaped display and the divided-rectangle display, which are in a sense isomorphic, are equally unamenable to efficient use of the display space. Any effort to gain efficiency by distorting the display to compress sparse areas or expand dense ones is likely to incur a penalty by hiding relationships that undistorted graphics would present clearly.

9. Graphics is useful both in portraying the structure of a program or data set and in providing identifiers for things that to some extent resemble the things identified. But at the present time, in the present state of computer and display technology, those are two quite different areas, and graphics appears to be of more practical value, in the context of programs and programming, in portraying structure than in identifying.

10. For portraying structure, there appears to be no real competition for graphics, but one of the best schemes may be so familiar and so widely used as a way of structuring text that one may not think of it as graphics. That scheme is indentation, of course. Outline-style indentation (which is often referred to in the computer context as *pretty-printing*) is capable of portraying hierarchical (or tree) structures very effectively, even if the components that are placed in the indented structure are symbols that bear no pictorial relation to what they represent.

11. For identifying things, graphical signs (icons) have an advantage over symbols, of course, in that signs by definition resemble to some extent the things that they identify, whereas symbols do not. However, the simpler and more abstract an icon, the less clearly it resembles what it stands for, and the more complex and concrete an icon, the more difficult it is for present-day computers to deal with. Most native speakers of western languages know many more symbols than signs. There is little hope of getting the most Americans to learn a system of several hundred icons, let alone several thousand, in order just to improve his or her ability to command or control a computer.

12. The conclusion that graphics is better, in the computer context, for portraying structure than for identifying things, is pertinent, of course, to the competition, in the personal computer world, between icon-based systems of the kind mainly

used with the Macintosh computer and symbolic systems of the kind mainly used with the IBM PC. The icons of the Macintosh (and, earlier, the icons used at Xerox Palo Alto Reseach Center and in the Xerox Star) are or were fine for identifying frequently encountered actions of the operating system, but it is significant that there is not a large amount of application software than carries the "desk-top metaphor" down deeply into specific applications. Our conclusion is that icons have very significant potential advantages over symbols but that a large investment in learning is required of each person who would try to exploit the advantages fully. As a practical matter, symbols that people already know are going to win out in the short term over icons that people have to learn in applications that require more than a few hundred identifiers. Eventually, new generations of users will come along and learn iconic languages instead of or in addition to symbolic languages, and then the intrinsic advantages of icons as identifiers (including even dynamic or kinematic icons) will be exploited.

13. It must be noted that a complex icon is in principle capable of identifying a complex object and, at the same time, portraying the structure of the object and even identifying some of its parts. Even a fairly simple icon can identify an object and some of the properties of the object. Probably the potential of iconic representation of computer and software constructs lies in this little-explored area in which identification of objects and portrayal of their structure are linked. At the present time, however, identification of objects and portrayal of structure in any deeper sense than approximating their shape are largely disjunct. You cannot tell much about the structure of a file from a file icon, and you cannot identify a pretty-printed function by looking just at the indentation pattern.

5. A Graphical Programming System

5.1. GRAPPLE

The exploration of ideas about monitoring the interpretation or execution of programs to some extent discouraged us from making the construction of a program monitoring system the main undertaking of the project. The key discouraging factor, as indicated, was not seeing how to cope with the inherent complexity of significant programs and data sets. We made the decision, despite conflicting enthusiasm for the dramatic quality and pedagogical value of dynamic graphical presentation, to make the main product of the project a graphical programming rather than a graphical monitoring system. It is a system called *GRAPPLE*. It provides a context within which the user, a programmer, can construct and use MDL programs graphically. GRAPPLE supports about 90 percent of the built-in operators of MDL and about two thirds of the built-in data types pertinent to applications of MDL. However, GRAPPLE does not deal with the many esoteric data types that relate to system programming, and it appears best to keep it that way because there is plenty for the user to learn as it is.

The user of GRAPPLE must know something about programming and something about MDL or LISP. We think that GRAPPLE may be most helpful, however, to a relative newcomer to programming who is studying a LISP-like language. The main value of GRAPPLE to the neophyte programmer probably lies in the concreteness of the graphical representation. Each datum has a residence site, and each use of a datum is connected by a line to the residence site of the datum. It is obvious, therefore, when two references are made to the same datum object, that just one object is involved, not two.

This section will provide a brief word-description of GRAPPLE. The description will not serve as a user's manual: a short user's manual has been built into GRAPPLE, and there is some prospect that a person who knows some LISP or MDL will be able to get started with only five or ten minutes' instruction plus the built-in user's manual (or tutorial). Nor will the description serve to support clear visualization of the diagrams presented by GRAPPLE. The aim here is just to convey the general idea. To go beyond that, there is no substitute for actually interacting with the program.

5.2. GRAPPLE's Initial Menu

GRAPPLE is a program. It exists as source language files and as what in MDL is called a *save* file of partly executable, partly interpretable code. We assume that a programmer will carry out a multi-session programming project with a *save* file and its descendents. At the beginning of the first session, the programmer loads the initial *save* file, say `grapple.save`, into a Digital Equipment Corporation VAX computer (with BBN Computer Co. BitGraph terminal) operating under the Berkeley Unix 4.2 or 4.3, by typing `mudsub grapple.save` and then a line-feed. (Although both programs can deal with both cases, Unix mainly uses lowercase letters, and MDL uses mainly uppercase letters.) Then, when the character string "RESTORED" appears on the screen, the MDL interpreter is running. The programmer starts GRAPPLE by typing <G> and a character that we shall call the *DO-IT* character. For the VAX computer with BitGraph terminal, the *DO-IT* character is the *line-feed* character.

After a short interval during which GRAPPLE loads a special font of characters, GRAPPLE presents the initial menu. This menu provides five alternatives. If the programmer is using GRAPPLE for the first time, he or she should select alternative 0, which will first describe the layout of the GRAPPLE screen and then initialize GRAPPLE for a series of programming operations. If the programmer knows the screen layout but is embarking upon a new series of of programming operations (or a single operation that will stand entirely by itself), the programmer should select alternative 1. If the programmer is resuming work and wants to begin a subseries of programming operations with a clean slate insofar as global variables are concerned, he or she should select alternative 2. If the programmer is resuming work and wants to begin a new function or macro or other top-level datum object, he or she should select alternative 3. And, if the programmer is resuming work broken off in the middle of the preparation of a function or macro or top-level datum object and wants to add a new component (and is very familiar with the system -- otherwise avoid this option), he or she should select alternative 4. Thus the first two of the five alternatives initiate an over-all programming venture and differ merely in showing or not showing a diagram of the screen layout. The last four of the five alternatives differ in respect of the amount of initialization action they take, and the last three of the five are for partial reinitialization when resuming work. Alternative 1 (as, also, alternative 0) initializes everything that can be initialized. Alternative 2 does not initialize the menu of operators, does not make you lose access to what you have programmed thus far in a series. Alternative 3 does not even initialize the situation with respect to global variables. Alternative 4 does not even initialize the situation with respect to function and macro parameters and local variables. Note that *initializing the situation with respect to variables* is different from *initializing variables*. When the situation with respect to global or

to local and parametric variables is initialized, the residence sites for that kind of variable are cleared They cease to be residence sites, and the names by which the variables were known to GRAPPLE cease to serve, until rebound to values, as the names of variables. When variables are initialized during the execution or interpretation of a program, in contrast, the variables are given values or, more precisely, values are bound to the names of the variables or values are inserted into the locations allocated to the variables.

5.3. Programming Sessions and the Save File

A programming session is a period of time during which the programmer is seated, presumably at the console or terminal. The programmer is free to break his or her course of interaction into sessions however he or she pleases. It is assumed that, at the end of each session within a series of programming operations, the programmer will create a new *save* file by selecting the item Save in a menu of general operations shortly to be discussed. Then, at the beginning of the next session, the programmer will restore that *save* file by using the mudsub command to the Unix operating system. Thus the series of *save* files (of which earlier members may be discarded) will accumulate the results of the programming operations, and the whole product of the (multi-session) series will reside in the final *save* file. There are, of course, facilities for filing away the results of programming as they are achieved. The product of the series may therefore exist redundantly, in the final *save* file and in various text files made in response to specific instructions from the programmer.

5.4. Compiling

GRAPPLE does not deal with compiling. It creates MDL source-language objects. Objects such as functions and macros may be compiled later on a VAX or DECSYSTEM-20 computer with the MDL compiler called *MIMC* for *Machine Independent MDL Compiler* despite the fact that running on only two different machines does not make it very machine independent.

5.5. The Screen Layout

GRAPPLE responds to the selection of any item from its initial menu by laying out the display screen in a pattern that is maintained, in its essentials, throughout a programming series. The layout pattern is explained briefly by the action of GRAPPLE in response to the selection of alternative 0. The layout has 9 main parts. (It would be better if it could be made simpler without leaving anything out.)

The center of action and attention for the GRAPPLE programmer is a large square in the central part of the upper half of the screen. At the outset, this square represents a general

MDL object. The task of the programmer is to give structure and content to the square by issuing commands, mainly with the mouse, that specialize the square into the GRAPPLE represention of the program (or data set) the programmer has (more or less clearly) in mind.

To the extreme left of the central square is a tall, narrow menu of 32 general programming operations. It is called the *main* menu. The first item in the main menu is `?Main`, and by selecting it and then selecting any other item, you can have GRAPPLE print out a brief explanation of the other item. (That sentence is still correct if you remove the two instances of *other*.) We shall return to more detailed examination of the main menu.

In between the main menu and the central programming square is the residence area for global data -- not for the built-in operators or programmed operators of MDL, but for global data to be used as data by programs under construction.

To the extreme right of the central square is a tall, narrow menu of 32 items, most of which are icons representing data types and classes. (Although 12 of the items are not icons representing either data types or data classes, the menu is called the *menu of data types and classes* or `DtC`, and selecting `?DtC` in the main menu and then an icon in `DtC` menu will get you a brief explanation of the icon.

In between the `DtC` menu and the central programming square is the residence area for function or macro parameters and local variables to be used by programs under construction.

Disposed horizontally just beneath the five areas just described, there is an area (the *shelf*) for miniature copies of program structures prepared by the programmer. Selecting `Down` from the main menu will create a miniature copy of the body of a recently completed function or macro and send it down to the shelf, and selecting `Up` will expand a miniature copy of a structure that is on the shelf and put it back up in the main programming square.

Just below the shelf is a menu of operators (the `R` menu) that can be thought of as three menus: a menu of built-in operators, a place to put auxiliary menus, and a menu of auxiliary menus. Together, they comprise eight lines. The first three lines are filled with short or abbreviated names of the most frequently used built-in MDL operators. The next four lines are vacant, waiting for submenus to be loaded by the programmer. The last line is the menu of auxiliary menus. Selecting `?R` from the main menu and then the short or

abbreviated name of a built-in operator confirms the name if it is the full name though short or yields the expansion of the name if it is abbreviated. ?R does not work for other names (i.e., names of auxiliary operators) in the R menu.

Just below the menu of operators is a row of four mode indicators. These tell the programmer whether or not the present situation calls for *programmer's initiative control* or *programmer's responsive control*, what kind of action (if any) the program is expecting the programmer to take, and what kind of action was taken the last time action was taken.

Finally, there is a fairly large area in which the programmer and GRAPPLE communicate in text. GRAPPLE puts out quite a lot of text, telling the programmer (more verbosely than the mode indicators) what the situation is and what GRAPPLE expects him to do. Sometimes, in its present state, GRAPPLE displays text faster than anyone can read it. When it does that, it is usually trying to figure something out about the compatibility of data types and classes and the printout is somewhat like subvocalization. The programmer does not need to know, in those cases, what he cannot read. When the programmer is called upon to respond by typing something on the keyboard, his or her response is echoed in this text area.

5.6. Programming a Very Simple Function with GRAPPLE

Before going into more detail about the various areas of the GRAPPLE display screen, let us exercise GRAPPLE by programming a very simple function. Let us prepare a function, which we shall call sq, which will return the square of its one argument, which must be a *real integer* or a *real continuum number*. (This example, and the sum-of-squares example that comes later, are dealt with more fully and more effectively in the user's manual that is built into GRAPPLE.)

We have loaded a GRAPPLE *save* file, we have typed <G>, we are looking at the screen layout described, we have a keyboard before us, and we have a mouse in hand.

In the central programming area, the initial display is a large square (a square rectangle -- no connection with the fact that the function we are programming is going to square numbers). The square is the simplest example of a category of figures we shall call *picts*, which are drawn with lines, to distinguish them (from here on) from the small pictures composed with FONTED, which we shall continue to refer to as *icons*. In the square are three icons, one in the middle signifying *general object*, which is what the square represents, one on the left signifying *applier*, which is what we want the body of our squaring function to be,

and one on the right signifying *general datum*, in which we are less interested right now.

It is a general rule of GRAPPLE that, if there is an icon in the middle of a pict, the icon signifies something about the nature of the pict. Thus there is a *general object* icon at the center of the square that represents a *general object*.

Not being very sure about what to do, we select the middle icon, the one at the center of the square. Selecting it does no more than to cause GRAPPLE to display some information and suggest that we select the left-hand icon, which is what an experienced GRAPPLE programmer would have done at the outset. We then select the left-hand (*applier*) icon with the mouse, moving the mouse in such a way as to make the mouse cursor (a +-shaped figure that moves about the screen as the mouse moves about the desktop) come close to the *applier* icon and then pressing the middle button of the mouse. The mode indicators and text in the text area thereupon ask us to confirm or disconfirm the selection. Another general rule of GRAPPLE is that each step involving selection with the mouse is followed by an opportunity to confirm or disconfirm. The left-hand button of the mouse confirms; the right-hand button of the mouse disconfirms. We press the left-hand button and, thereupon, the square representing the *general object* turns into, or is replaced by, a square that is divided into two rectangles by a vertical midline. It represents an *applier*, and in the middle of it, superposed upon the midline, is an *applier* icon, a small rectangle with a line down the middle. To the left of the *applier* icon is an *operator* icon, and to the right is an *operand* icon.

We are programming a squaring function, and we are now at the point of selecting the operator for the one and only expression of the body of the function. We therefore select the left-hand icon, the *operator* icon, and confirm the selection. The mode indicators and text area then ask us to select the operator from the operator menu. It accepts the mouse cursor, and we position the cursor over the operator *.

When instructed to use the operator *, GRAPPLE has to ask a supplementary question. The multiplication operator of MDL can accept any number of operands, and GRAPPLE needs to know how many operands there will be. It therefore asks us, in the text area, and asks us to respond via the keyboard and to terminate the response with the DO-IT character. Inasmuch as we want a function that will square a number, we say that there will be 2 operands. After a short interval of computation, a rectangle slightly smaller than the left-hand half of the divided-down-the-middle applier rectangle fits itself into the left-hand half to represent the operator. In the middle of this rectangle is an *operator* icon and

to the upper left is a * label. At the same time, two smaller rectangles, which just happen to be squares, fit themselves, one above the other, into the right-hand half of the *applier* square. They represent the two operands. The *applier* icon is still in the center of the *appler* square. An *operator* icon is in the center of the *operator* rectangle. And an *operand* icon is in the center of each *operand* square. To the upper left of each *operand* icon is an *applier* icon, for use in the event that the programmer decides to develop the operand as the result of applying something to something, and to the lower right is a datum icon for use in the event that the programmer decides to supply a datum directly, either as a literal datum or as a parameter or local or global value. In the current case, the datum icon is a *real* icon because the multiplication operator works only with data of type *real integer* or type *real continuum number*, i.e., of class *real*.

We want to supply the same number to each of the two use sites, i.e., to the location of the *real* icon in each of the two *operand* boxes. We want this number to be the one argument of the function we are defining. We begin with the upper *operand* box and, with the mouse, select its *real* icon and confirm the selection. Then we signal, with the left-hand and right-hand mouse buttons, that we want a nonliteral datum and that we want a new datum. GRAPPLE then asks us to choose a residence site for the datum. We select a residence site in the datum area to the right of the central programming square -- in the left-hand column of that area, which is labeled R for *required parameter*. Then GRAPPLE asks us to choose a data type or class. The DtC menu accepts the mouse cursor, and we choose and confirm the *real* icon. GRAPPLE checks that *real* meets the requirement of the program under construction (which it does, trivially, inasmuch as *real* was the class required, and then GRAPPLE asks us to supply an illustrative value for the parameter. We type 7, which is of type *real integer* and therefore of the class *real*. The 7 takes up its abode at the selected residence site, underneath a *real* icon, and a spline forms, connecting the residence site to the use site.

For the other operand, we want to use the same required argument. We select the *real* icon in the second (lower) *operand* box and confirm it. Then we signal, with two clicks of the right-hand mouse button, that we want a nonliteral datum and that we want to use an old one, already set up at a residence site. Then we select the residence site of the 7 by moving the mouse cursor to the 7, and we confirm the location. Thereupon, a second spline forms, connecting that residence site to the second use site.

That completes the programming of the body of the function. All we have to do to complete the function definition is to signal to GRAPPLE that we have completed our part.

At this point, however, we pause to check the operation of the body of the function upon the illustrative datum we supplied, the 7. We select and confirm Eval from the main menu, and GRAPPLE responds by printing 49 in the text area. Then, to indicate that we are finished programming the function, we select and confirm FiniF from the main menu. GRAPPLE first shows us the result of processing the most recently programmed expression and packaging it as a function. (This is useful when one is programming a multi-expression function, but it is a bit redundant when, as currently, one is programming a function that will have as its body a single expression.) Then GRAPPLE asks for a name for the whole function. We type sq on the keyboard and terminate it with the DO-IT character. GRAPPLE then displays the MDL text of the new function definition:

```
<DEFINE sq (r.1) <* .r.1 .r.1>>
```

We foresee that this little function will be useful, so we select and confirm Down and then File and then MuGrP from the main menu. Down puts a miniature copy of the new function (with its name above it as a label) on the shelf. File files the new function, together with all the paraphernalia for displaying it graphically, away in a Unix file, and MuGrP puts the name sq into the first unused slot in the operator menu and prepares an entry for the data base that describes operators.

Even though the foregoing yielded only a trivial program, it illustrated the use of the central programming area, the main menu, the R menu, and the DtC menu. Let us turn, now, to fuller examination of those menus. Then we will use the function sq in programming a function that will sum the squares of the numbers in a structure of numbers.

5.7. The Main Programming Operations of GRAPPLE -- The Main Menu

A programmer using GRAPPLE intends to prepare a MDL program, and a MDL program typically consists of functions and data sets, sometimes also of macros. Usually, there are more functions than data sets or macros, and GRAPPLE assumes, unless it is told otherwise, that what the programmer wants to do when he gets a fresh GRAPPLE or completes something by selecting item FiniF from the main menu is to prepare a MDL function. In the course of creating a program, however, there are quite a few other things to do, and quite a few specific things to tell the programming system. It will serve as a general introduction to GRAPPLE if we examine briefly a set of these other things that are dealt with through the main menu.

We have already dealt with the first three items of the main menu, which provide expansions or explanations of items in menus. Here, briefly, is what the other 29 items do:

Chkpt	Let the programmer record the state of GRAPPLE at a particular point in time (colloquially, *create a checkpoint*) or go back to a state *checkpointed* earlier.
Decl	Insert declarations into the argument list of the function or macro most recently created.
Down	Make a miniature copy of the most recently prepared function or macro and put it down on the shelf.
Embed	Select a location in the object that is in the central programming square. Select a miniature structure from the shelf. Select a top-level expression from the body of the function or macro represented by the miniature structure. Embed that expression in the object in the central programming square at the point selected.
Erase	Erase the central programming area, the main menu, and the residence sites for global variables, function and macro parameters, and local variables.
ErrRt	Error return. Return to the top level of the MDL interpreter. (For use after an error condition has arisen.)
Eval	Evaluate the expression most recently programmed. This evaluation uses as the values of parameters the values provided as *illustrative values* or *default values* or *local values* as the expression was being programmed. `Eval` ceases to be capable of evaluating an expression at the time the programmer declares, by selecting `FiniF` or `FiniX`, that he is finished with the expression and wants to proceed.
File	File, using a filename that is the name of the function or macro plus the extension `.mud`, the function or macro most recently programmed, together with its tree and all its hieros. (The programmer need not be concerned with the tree or the hieros, or even what they are.)
Find	Find the presently displayed or non-displayed menu in which a specified operator or specified data type or class is located (and its index in that menu).
FiniF	Finish function. The programmer has completed the body of a function or macro, so package the body and the argument list into a function or a macro and get a name for the new function from the programmer.
FiniX	Finish expression. The programmer has completed a top-level expression, so rearrange the display in the central programming area in preparation for the programming of another top-level expression.

Glb	Create a global datum with a residence site in the global data area.
Glbs	If the global data have been removed from the screen, return them to their residence sites and redisplay them.
Icons	If the miniature copies of programmer-defined functions and macros have been removed from the shelf, redraw them in their old locations.
Init	Present a menu for initialization that is like GRAPPLE's initial menu except that it works with the mouse and includes, in addition to the five alternatives of the initial menu, three special reinitialization alternatives. They are: (1) to reinitialize the mechanism for deferred definition, i.e., for forward reference to a function or macro not yet defined; (2) to reinitialize the situation as regards global data; and (3) to reinitialize the arrangement that keeps track of the component diagrams that have been used and prevents confusion between what has been used and what GRAPPLE will create to represent new programming strucutres. The third special reinitialization is drastic: in effect, it destroys all the graphic structures that have been defined thus far.
Macro	Give the programmer a chance to make the next-constructed top-level object be a macro instead of a function (or to change his or her mind and go back to the default, which is to create a function).
MuGrP	To the menu of operators from graphical programming. Put the name of the function or macro most recently defined by graphical programming into the menu of operators (the R menu) and create a data-base entry for it.
MuMDL	To the menu of operators from MDL. Put the name of a specified MDL object into the menu of operators and create a data-base entry for it.
More	Disregard the fact that the programmer has indicated that he or she is finished with the programming of a function or macro and prepare to add another top-level component to the function or macro.
New-t	Prepare to define a new data type.
Ratch	Ratchet. Ordinarily, the data base of operators that GRAPPLE knows about is initialized, *via* initialization 0 or initialization 1, by being copied from a data base called the *initial data base*. Selecting the item Ratchet replaces the initial data base with a copy of the currently working data base, which contains not only the information from the initial data base but also information about functions and macros the programmer has constructed. Ratchet thus updates the inital data base to include the newly programmed objects. The only way to get back to the "initial initial" data base after ratcheting is to load a file. The file can be either an old *save* file that contains the "initial initial" data base (which returns the whole graphical programming system to the state recorded by the *saving*) or a text file that contains a copy of the "initial initial" data base. The file `initial-grapple-data-base.mud` contains a copy.

ReDrw	ReDraw the program diagram corresponding to the current program tree. This assumes that a hiatus has been introduced by the use of the Erase menu item.
Res-t	Result type. Accept the designation of a data type or class and then ensure that the next function or macro constructed by graphical programming will return an object of that type or class.
Retrv	Retrieve. Retrieve the contents of a Unix file, the name of which is to be specified. In MDL terms, the contents will be FLOADed into the MDL interpreter. If the retrieved file is one created by the File menu item, then it should be a file of information from the present programming series. Otherwise, some of the structures in the file may overwrite structures in the present system.
Save	Save -- i.e., checkpoint, i.e., record the current state of the system -- in a MDL *save* file with the filename that the user will specify (and with the extension .save). This operation takes about a minute.
Sgmnt	Give the programmer a chance to make the next object to be constructed by graphical programming be a MDL segment (consisting of the contents of a structured object without the container) instead of (if indeed it is going to have contents) a full-fledged structured object. (Also give the programmer a chance to make the system revert to the default state, which is to keep the containers of structured objects.)
Tree	Erase what is currently displayed on the screen and display the *program tree* that contains all information about both the expressions of the function or macro that is under construction and the graphics used to represent them. (The programmer can stop the display of the tree, which will occupy several screenfuls, by holding down the *control* key and pressing the *G* key and then can restart the display of the tree by typing <ERRET T> and then the DO-IT character.) When the tree has been displayed, GRAPPLE erases it and redisplays what was on the screen before. This mechanism is, of course, for programmers who understand the tree structure and something about how GRAPPLE does what it does.
Up	Erase what is in the central programming area and draw there a full-sized copy of a miniature structure selected from the shelf.
tCChk	Type and Class Checking. Turn *on* or *off* the mechanism that GRAPPLE uses to make sure that the data types and classes specified in the program under construction are compatible with the operators that are specified. Even when this checking mechanism is turned off, the basic type-checking mechanism of MDL will be operating -- unless it, too, has *been turned off.* To turn it *off,* type <DECL-CHECK %<>> and press the DO-IT key. To turn it *on,* type <DECL-CHECK T> and press the DO-IT key.

5.8. Operators and the Menu of Operators

The operator menu is used during the construction of a function or macro to identify component operators.

GRAPPLE is in a sense both a language and a programming system. Both the language and the system are graphical, but in both cases the graphics are diluted with some alphanumeric text. For instance, most of the names of built-in MDL operators are text names. And that fact immediately raises a problem that makes GRAPPLE harder to learn than it should be. The problem is that GRAPPLE needs, or at any rate tries, to present a lot of operator names in a menu simultaneously, between 72 and 144 of them, and that imposes a requirement for short names. GRAPPLE's names for operators are therefore usually abbreviations of MDL's names for the same operators. At present, the abbreviations are 1, 2, or 3 letters long. We recognize that such short abbreviations cannot cope with large sets of operators and that it would be very helpful to have on tap a few thousand operators. One of the next changes will be, therefore, to raise the limit on the length of operator names, probably to seven for programmer-defined operators and infrequently used built-in MDL operators.

But let us describe what exists now: As indicated in the introductory description, the menu of operators can be viewed as having three parts. The first of them (three horizontal lines) contains three special menu labels and short or abbreviated names of 69 built-in MDL operators.

The three special labels are +++, XIT, and DFR. +++ calls for additional, supplementary lines of menu. XIT is useful if your mouse is in the operator menu and you would like to get out without selecting an operator. DFR is useful if you want to refer to a function or macro that is not represented in the menu -- and may not even exist yet, as is the case during the definition of a recursive function.

The 69 operators with short or abbreviated names are in part self-explanatory (to people who know MDL or LISP), in part not. Explanatory expansions, in most cases MDL equivalents, are shown in parentheses:

& (AND)	*	+	-	/
0?	1?	= (SET)	== (SETG)	==? (IDENT?)
=? (EQUAL?)	A (APPLY)	ASC (ASCII)	ATN (ATAN)	BNL (CR-LF)
C (STRING)	CMP (STRCOMP)	CND (COND)	COS	CØ (PNAME)
EM? (EMPTY?)	EXP	G=? (GR-EQ?)	G? (GR?)	I-C (ISTRING)
I-L (ILIST)	I-U (IUVECTOR)	I-V (IVECTOR)	L (LIST)	L=? (LS-EQ?)
L? (LS?)	LN (LOG)	LNG (LENGTH)	MAX	MEM (MEMBER)

```
MIN             MOD             MP1 (MAPF)      MPL (MAPLEAVE)  MPR (MAPR)
MPS (MAPSTOP)   MP_  (MAPRET)   PAR (PARSE)     PR (PRINC)      PRG (PROG)
PRI (PRIN1)     PRT (PRINT)     RD (READ)       RPT (REPEAT)    RST (REST)
RTN (RETURN)    SIN             SQT (SQRT)      TYI             U (UVECTOR)
UNP (UNPARSE)   V (VECTOR)      VAL (GVAL)      X? (STRUCTURED?)
f (FLOAT)       i (FIX)         1TH (NTH)       m? (MONAD?)     mem (MEMQ)
or (OR)         t (TYPE)        t? (TYPE?)      q (QUOTE)       val (LVAL)
```

(The mnemonic that makes MP_ a deviously reasonable abbreviation for MAPRET is that ASCII character number 95 (decimal), which now usually prints as an underbar, used to print usually as a back arrow, and back arrow is not too far from the meaning of MAPRET: send something back to the place where the result of this mapping operation is being accumulated.)

Below the four vacant lines is a line of labels that stand for submenus. It seems best, from the present point of view, to replace the contents of this line with the letters of the alphabet and the decimal digits. There would then be a submenu for each letter and each digit.

5.9. Data Types and Classes

Each object in MDL is a member of a category of objects called a *type*. The term *class* is not a MDL term, but it is used in GRAPPLE in the way to be described. It is used in other languages in other senses than the present one. Because there are not enough terms like *type*, *class*, *set*, *category*, and *ilk* to go around, GRAPPLE finds itself using *class* to stand for a group of objects that includes members of more than one type. Perhaps the prototypical example is the class *real* which includes *real fixes* and *real floats*. (This matter is difficult to discuss in a computer context because most computer languages speak of *integers* as separate and distinct from *real numbers*, whereas of course real integers are just as real as real fractions. At the present time, GRAPPLE is quite as confused as the rest, setting *integers* and *continuum numbers* into contradistinction despite the fact that, mathematically speaking, real integers reside on the real continuum along with all the fractions and despite the fact that computers do not have the denumerably infinite storage capacity to represent integers, let alone the nondenumerably infinite storage capacity to represent continuum numbers.) In any event, GRAPPLE distinguishes classes of objects as well as types of objects. Any MDL object at all is a member of the class O, or general object. Any MDL object that has structure (i.e., is capable of containing members) is a member of class X, or structure. And any MDL object without structure, i.e., any elementary or unstructured object, is a member of class o, or element.

In GRAPPLE, an effort is made to use icons instead of letters such as i (for the type *real integer*), z (for the type *real continuum number*), r for the class *real number*, L for the type *List*, and C for the type *Character string*. It is not feasible, however, to print the icons in a report such as this. It will have to suffice to describe them, and indeed most of the graphics, as though the reader were looking at the display screen while programming.

Icons for the most frequently used data types and classes are presented in the menu of data types and classes in the upper right-hand part of the screen. There are 32 items in the columnar menu, of which 20 denote data types and classes. The first two icons resemble the first two items in the menu of operators. They are a +++ sign and an XIT sign, and their functions are analogous to the corresponding functions in the operator menu. The last 10 items are the decimal digits. They are used sometimes as identifiers of auxiliary sets of data types and classes and sometimes just as integers to be selected by the programmer as input to GRAPPLE. There are two ways to give integers to GRAPPLE: with the mouse, using the single-digit integers in the DtC menu, and with numeric keys in the upper row of the main keyboard (but not in the numeric keypad). The rule that (except in one regretted case, which is identified by the displayed instructions) governs which way to specify integers is: use the menu digits if it is not possible, in the local context, for more than one digit to be needed to constitute a single input integer; otherwise use the upper row of the keyboard to type as many digits as needed and then terminate entry of the series of digits (which may of course be a series of just one digit) with the DO-IT character.

The icons representing data types and classes are special characters designed on a matrix 10 pixels high and 16 pixels wide. From top to bottom, they represent (3) general object, (4) general datum (equivalent to general object because any MDL object can be treated as a datum), (5) operator, (6) monad (if structured, empty; otherwise elementary), (7) element, (8) atom, (9) byte, (10) character, (11) real, (12) real integer, (13) real continuum number, (14) structure, (15) list, (16) vector, (17) uniform (vector) (MDL UVECTOR), (18) character string (MDL STRING), (19) byte string (MDL BYTES), (20) false, (21) applier (MDL FORM), and (22) function.

If the desired data type or class is not in the permanent menu of data types and classes, select the top item, which resembles +++, and then select the appropriate one of the integers, the one that identifies the auxiliary set of data types and classes that includes the one you want. At the present time, only the 0 is used for additional data types and classes, so there is no problem in remembering in which set a particular auxiliary data type or class is located. For the hypothetical future, when there may be more data types and classes,

the item Find in the main menu will help you to find the correct submenu.

The auxiliary set (0) of data types and classes at present includes the following data types and classes (with the GRAPPLE name first, the MDL name, if any, or otherwise the expanded GRAPPLE name, second):

1. reserved for Z, FRAME,
2. no longer used, was MDL ENVIRONMENT,
3. Y, ASSOCIATION,
4. reserved for N, Number,
5. reserved for K, Complex,
6. P, POINT,
7. reserved for Q, QUANTITY,
8. n, scalar,
9. f, continuum,
10. h, integer,
11. u, reserved for imaginary,
12. y, reserved for continuum imaginary,
13. reserved for integer imaginary,
14. k, index,
15. reserved for S, SET,
16. W, TUPLE,
17. I, Uniform of integer reals (UVECTOR of FIXes),
18. T, TEMPLATE (also used to represent 'T' for 'true'),
19. reserved for d, Date,
20. reserved for t, Time,
21. no longer used, was H, HANDLER (of interrupts),
22. J, MACRO,
23. M, MSUBR,

24. !, Local value (identified in MDL by prefixed period),

25. g, Global value (identified in MDL by prefixed comma),

26. s, SEGMENT,

27. =, CHANNEL,

28. unused,

29. unused,

30. unused,

31. unused,

32. unused.

The unused icons are available for use by the programmer. If the programmer wants to substitute other icons for the unused ones, that is fairly easy to do with the FONTED package.

5.10. Programming a Function to Determine the Sum of the Squares

The function that we now wish to program, let us assume, is one that will compute the sum of the squares of the numbers in a structure (e.g., a list, a vector, or a uniform (i.e., a MDL UVECTOR)) that will be provided to the function as its one and only argument. We have a function, sq, that will square individual *real* numbers, and we intend to use it as a component of the new function.

We need to reinitialize GRAPPLE. We therefore select and confirm Init from the main menu. GRAPPLE presents a menu that is similar to the initial menu but offers three additional alternatives. Since we want to begin on a new function, it is appropriate to select initialization level 3, for FUNCTION. GRAPPLE erases parts of the display and redraws parts. Again we see the *general object* square in the central programming area. There were no global variables before, and there are none now. The slate of parameters and local variables has been wiped clean.

The strategy we adopt for determining the sum of squares is a simple one based on the use of one of MDL's mapping operators. The operator MAPF (abbreviated MP1), when given three arguments, applies its first argument to the accumulated results of applying its second argument to each member of its third argument. So, if its first argument is the addition operator, its second argument is a squaring operator, and its third argument is a

structure of *real* numbers, it will return the sum of the squares of the real numbers, which will, of course be a real number. It will be a *real integer* if all the members of the structure are *real integers*, and it will be a *real continuum number* if any one or more of the members of the structure is or are of type *real continuum number* (and the rest, if any, are of type *real integer*).

Pursuing that strategy, we select and confirm the *applier* icon from the *general object* square. GRAPPLE substitutes an *applier* square, with the now-familiar line down the middle. We select and confirm the operator icon, and GRAPPLE asks us to specify the operator. We select and confirm MP1 from the operator menu, and GRAPPLE asks us how many operands our use of the operator will take. (If we are unsure, it is all right to overestimate, but not to underestimate.) We respond from the keyboard, as requested to, with 3. GRAPPLE fits an *operator* rectangle (with the label MP1) into the left-hand half of the *applier* and three *operand* rectangles into the right-hand half. These *operand* rectangles have *operand* icons at their centers, of course, and *applier* icons in their upper left quadrants, and datum icons in their lower right quadrants. The datum icon in the box representing the first argument is actually a list of two icons, indicating that the object whose residence site is connected to that use site must be either a *false* or an *operator*. The datum icon in the second argument box is the *function* icon. And the datum icon in the third argument box is the *structure* icon.

We begin with the first argument. We want to accumulate the result with the + operator, so we select and confirm the datum icon of the first argument box, signal to GRAPPLE that we want a literal datum, and type ,+ on the keyboard and terminate it with the DO-IT character. The comma followed by the plus sign represents an object of type *VAL* (MDL GVAL). The value of the *VAL* will be the addition operator, since the built-in operators of MDL are bound as global values to the atoms that are their names. The label ,+ replaces the datum icon of the first argument box.

> The procedure just outlined involves using a bit of syntax -- the prefixed comma to specify 'global value of' -- that is essentially MDL and not GRAPPLE. A way of specifying the operator associated with a name without using that bit of MDL syntax has been programmed into GRAPPLE, but let us make do with the comma notation here.

Turning to the second argument with the aim of making it be the squaring function already prepared, we select and confirm the datum icon, which is the *function* icon, of the second argument box, and we signal to GRAPPLE (with the mouse's left-hand button) that we want a literal datum again. We type ,sq on the keyboard and terminate with the DO-IT character. (It turns out that we shall not in this case use the menu item sq that we put

into the operator menu with MuGrP. Inasmuch as we are using the squaring operator as an operand of the mapping operator, it is appropriate to specify it by one of the regular means of specifying operands.) The label ,sq replaces the datum icon of the second argument box.

Turning to the third argument with the aim of making it be a structure of numbers that will be the one parameter of the sum-of-squares function, we select and confirm the datum icon of the third argument box and signal GRAPPLE, by clicking the right-hand mouse button and then the left-hand mouse button that we want a datum that is nonliteral and new. (Incidentally, GRAPPLE is providing rather detailed instructions, just in case we have not performed the procedure so often that it has become rote.) Then, again in response to instructions, we select a residence site for the third operand of the MP1 operator. We do so by chosing and confirming a location (any location) in the R column of the residence area to the right of the central programming square. Then GRAPPLE asks us to choose a data type or class. In order to make the function as generally applicable as possible, we choose the class *structure*. We note that there is a problem here, that the class *structure* is even too general for what we want, which is a structure that contains only *real* numbers, and we resolve to keep in mind that trouble could arise if, for example, we tried to apply our new function to a string of characters. But we do not want to divert now from our immediate task. So we confirm the selection of *structure* and type, as an illustrative value, (1 2 3). A spline forms, connecting the residence site of the (1 2 3) to the use site in the location of the datum icon in the third argument box.

Before declaring that we are finished, we evaluate the expression, using the illustrative datum, by selecting and confirming Eval in the main menu. GRAPPLE types the result -- 14 -- of the evaluation in the text area. Seeing that the result is correct, we select and confirm FinlF from the main menu and name the new function sos. For future reference, we select and confirm, also, Down, File, and MuGrP.

Finally, not being fully satisfied with the single test afforded by the illustrative value (1 2 3), we test sos further by using it as an ordinary MDL function, which it is. We use the text area to type to the MDL interpreter, which is listening at all the times when the first mode indicator indicates PROG'S INITIATIVE.

```
<sos (3 4 5)>
50

<sos [1 2 3 4 5]>
55

<sos ![100 -100]>
```

20000

And so on. (But note that this testing has been done symbolically in the language MDL, not graphically in the language GRAPPLE.)

6. A Tutorial User's Manual

If a concept demonstration system such as GRAPPLE is going to be examined by people outside the immediate developing group, the system must have some kind of instructions for use, some kind of user's manual. The creation of such a manual, however, is a considerable task, a task that could escalate into a project comparable to the development of the demonstration system, itself. The approach we have taken to the preparation of a user's manual was determined mainly by the fact that it is very difficult, or impossible, to deal effectively with continually changing displays in a static medium such as print on paper. We have built most of the user's manual into GRAPPLE. That is not to say that the user's manual is *designed into* GRAPPLE. GRAPPLE was substantially completed when we began to consider how to incorporate the user's manual. Nevertheless, the user's manual does have a 'sense' of what the user is trying to do, or should be trying to do, and it does put the instruction and advice into close association with the particular items of interaction to which they pertain.

The user's manual has two main parts: (1) 'Ink-and-Paper Part of GRAPPLE User's Manual' and (2) the part that is built into the program GRAPPLE. Except for an eight-page list of abbreviations, acronyms, and expansions, the ink-on-paper part is just seven pages long. The user should read it first. It deals mainly with how to get GRAPPLE started, how to turn on the user's manual, and what the main limitations are. Almost all the information about how to use GRAPPLE, once it is running, is in the part of the user's manual that is built into GRAPPLE.

The part that is built into GRAPPLE consists, itself, of two parts. The main one of the two parts is based on nine exercises and provides instruction and explanation related to the exercises. The minor one of the two parts explains the items in the main menu and the menu of data types and classes whenever, in the process of using or creating a program, they are selected and confirmed.

Of the nine examples with which the major part of the built-in user's manual is concerned, the first two are examples of how to use existing operators to process data. The last seven are examples of programming which, within the context of MDL and LIST, is mainly a matter of defining functions (and turning the definitions into actual functional objects that can be applied to operands).

The major part, dealing with the nine examples, 'knows' which one of the examples the user has selected. At each one of various strategic locations within GRAPPLE, there is a code

that causes a digression to the user's manual if it is turned on. At each digression, a small data base is updated, so GRAPPLE will know which visit this is to this specific location, the first, the second, and so on. GRAPPLE then presents a file of information that is appropriate to that place visited for that (ordinal number) time. The information is displayed as a 'section' of one or more 'subsections', each subsection being short enough to fit into the text display at the bottom of GRAPPLE's screen layout. When he or she has finished looking at a subsection, the user must release it by pressing the left-hand or the right-hand button of the mouse. Indeed, the user must release each of the subsections (and thus release the section) before proceeding with his or her interaction with GRAPPLE.

The user's manual assumes some knowledge of LISP-like languages. The target is, essentially, a person who is taking a first course in programming with LISP as its vehicle or who is taking a more advanced, but not very advanced, course in LISP. We think that some people may be able to deal with GRAPPLE on the basis only of the user's manual. To be realistic, however, we should face it that a user's manual has to be debugged, quite as surely as a program, and that there will no doubt have to be some person-to-person communication to deal with problems not foreseen during preparation of the user's manual and bugs not revealed thus far in the code of GRAPPLE. With reference to the latter, let us say that the main parts of GRAPPLE have been used many hours during the last year without encountering bugs or glitches, but that there are so many lines of interaction off the main track that there has been no way to check them all exhaustively and there are bound to be unfound bugs in them.

7. Issues in Graphical Programming and Monitoring

We conclude this report with a discussion of issues that appear, in part on the basis of the experience provided by this project, to be the main present issues in the field of graphical programming and monitoring. We shall try not to duplicate much of what was said in the conclusions about graphical monitoring, but we shall touch upon most of the problems touched upon in our report of October 14, 1984.

The main issues appear to be:

1. Whom, if anyone, can graphical programming and/or monitoring help?
2. What are the good things that graphics can do for programming and for the understanding of programs?
3. What are the main limitations of present-day computer graphics that stand in the way of exploiting the potential benefits?
4. Is there a sufficient or an appropriate effort to exploit graphics as an aid to programming and/or understanding programs?

7.1. Whom Can Graphics Help?

As for who may be helpable, there appeared at the outset to be three main possible targets for graphical programming and monitoring: (1) experienced programmers, (2) inexperienced programmers, and (3) people who do not consider themselves to be programmers but who work with spreadsheets, data management systems, word processors, and/or other software that calls upon the user to direct the computer's actions on sets of numerical, graphical, or textual data. Recognizing the third group led us to name it and to restructure the breakdown: (1) experienced explicit programmers, (2) inexperienced explicit programmers, (3) experienced implicit programmers, and (4) inexperienced explicit programmers.

As for how they may be helped, it appeared at the outset that the main ways were to help them write programs and to help them understand programs written by others. Almost immediately, however, the scope was broadened to include the updating of programs, which involves writing but also understanding, and the use of programs, which does not quite involve writing programs and involves understanding what they do but not how they do it. So we think of (1) help in writing, (2) help in understanding, (3) help in updating, and (4) help in using programs.

The general conclusion in this area is time-dependent: Eventually, computer graphics will help all four kinds of people do all of the four kinds of things that they try to do. In the short term, limitations of the equipment for graphics will make it hardest to help the

people who do the most complex things and who are already good at doing them with the aid mainly of symbolic techniques. So group (1), experienced explicit programmers, will in the near term be the hardest to help; group (2), inexperienced explict programmers, will be easier to help; group (3), experienced implicit programmers, will be still easier to help; and group (4), inexperienced explicit programmers, will be easiest to help. Thus we think that a graphical spreadsheet system for newcomers to spreadsheets might be a fairly easy project with high payoff, whereas targeting more experienced users would complicate the problem and escalate the requirements without increasing the benefit proportionately, and trying to broaden the aspiration to embrace programming in nonrestricted domains would make things very complicated and difficult. Looking back, we consider the selection of GRAPPLE as a task to have been too much in the general purpose direction and too much oriented toward programming instead of use of programs. However, it seems possible to modulate GRAPPLE into a program-using aid without making it cease to be a programming system.

Perhaps the general conclusion, if there is one, is that fairly strong constraints make computer projects feasible, and that the constraints appropriate to the newcomer, as distinguished from the experienced programmer, and to the user, as distinguished from the writer or updater of programs, are needed, at the present time, to keep computer system developments from getting out of hand. Our considerable difficulty in getting promising handles on facilitation of the understanding of programs is just what one would expect from this point of view. There are infinitely many ways to misunderstand a program and, presumably, the user of an understanding aid starts out misunderstanding a program. The strategy one adopts at the outset of writing a program is in a sense a track, and it is easier to help something stay on a track than to help it find and get back on the right track in the presence of many wrong ones. This line of thought puts updating somewhere in between writing and understanding.

One of the main sources of difficulty in working with large programs is the problem of presenting in one viewing space all the parts of a program that are pertinent to a given event or action. In a good programming language, program text can be marvelously compact, and we do not see how to make a graphical representation that retains the information of the program text and that is, for the average program, more compact. Moreover, experienced explicit programmers have developed the skill of seeing the structure of a program "in the mind's eye" -- which is to say that, in a sense, they already use a graphical technique.

For inexperienced explicit programmers, we now believe, graphical programming and monitoring can be extremely helpful. The help can come from two different directions. First, graphics makes programming objects concrete and immediate, whereas symbolic representations tend to make them abstract and mediate. By *mediate*, we mean *not themselves immediately present, but referred to by name.* It is simply easier for people without much programming experience to manipulate present objects directly than to control absent objects by making patterns of their names. Second, a graphical programming system seems more likely than a symbolic programming system to build a model that it can understand and use of the program that the programmer is constructing. With such a model, the programming system can be of great help to the inexperienced explicit progrmmer -- can keep him or her out of several kinds of trouble and can even make helpful, constructive suggestions. The first kind of help is essentially graphical. The second is not, and it will surely be possible to realize some of its benefits in nongraphical systems. But we envision that it will be possible, with graphical programming and monitoring, to facilitate significantly both the learning and the performance of programming by inexperienced explicit programmers. It may take a display with several million pixels and a processor that executes several million instructions per second to do that, but that is almost exactly what it is feasible, now, to devote to the undertaking.

For the near term, the target groups easiest to help will be the experienced and the inexperienced implicit programmers. The inexperienced will be the easier of the two, but they may not be as willing to invest in the required aids, and it may be the experienced implicit programmers who actually receive the most help. In any event, implict programmers are a very important category, much larger than the category of explicit programmers, and usually much more closely involved in the activities that spell measurable profit and loss or measurable success or failure for their organizations. In any event, we are talking about most of the most creative and most productive users of computers.

The kind of graphical aid to the use of computers and software that we have in mind is, for one point of view, just an extension of the application of graphics, the "office metaphor", that has been popularized in connection with the Macintosh computer, i.e., the use of a mouse and icons to control actions of the operating system, the word processor, and so on. But the nature of the extension is important. One key dimension of it is the dimension of coherence or integration in the entire repertoire of software employed by the user. If the user deals with a motley collection of independent programs, then no amount of graphics is going to take the user much beyond the limits built into the individual programs. If the

user deals with an integrated family of precast programs that respect the group property of data sets (what is output from one function is input to another), and if systematic use of graphics is built into the programs, then it will be possible to carry something like the desk-top metaphor down to useful depth, and the graphics will be significantly helpful. But the real value of graphics will become evident when the user works with an integrated -- but also open-ended -- system of functions that run in an interpreter comparable to a LISP or MDL interpreter. (Think of the personal computer as a LISP machine.) Then there will be a continuum from (a) applying a major program to m files of data to (b) applying a built-in operator to n integers. The user will see that it is ridiculous to feel at home working with whole programs while fearing to enter the ominous zone of programming where one works with parts of programs. Graphics will help reduce everything to a common basis, and it will make the common basis concrete and "common sense". A good graphical programming and monitoring system will turn a lot of implicit programmers into explicit programmers, and that will substantially change the nature of software and its use. The educational value of graphics will make itself felt in offices, laboratories, and schools.

7.2. Some of the Good Things Graphics Can Do

We have mentioned various benefits at various points in this report, but it may be worthwhile to make a definite list of them:

1. Make programs, data sets, and the actions of programs on data sets more concrete and objective.

2. Give data a real presence. (In a symbolic program, one does not see the data; one just sees references to it, usually to the names of individual items and structures of data.)

3. Emphasize structure, both the structure of data and the structure of programs, and to some extent separate the display of structure from the display of content.

4. Provide several levels of abstraction, capable of dealing with corresponding levels of detail. (In conventional programming, there is no intermediate ground between the full detail of program or data structure plus content and the mere name of the program or data.)

5. Lay down constraints, especially syntactic constraints, to form a context within which the programmer or user of programs can work. By analogy, provide a map to the navigator and let him draw his course on it instead of on a blank piece of paper.

6. Mark the sources or locations of such crucial features as alterations of the flow of control, side effects, optional arguments, operands that are never evaluated. (In most conventional languages, these features have no distinctive appearance.)

7. Appeal to the spatial sense that is allegedly so well developed in our society as a result of extensive and intensive watching of television. (Or, to say much the same thing another way: Depend less on abstract symbolic representation, which, as an information processing skill, is not very well developed among relentless watchers of television.)

Substitute regular, diagramatic structures for irregular, textual structures, and take advantage of the eye's ability to detect deviations from regularity.

Avoid the difficulties posed by the fact that there are so many natural languages by developing much further the international system of signs.

7.3. *Limitations of Computer Graphics That Stand in the Way of Exploiting the Potential Benefits*

The graphical display and processing capabilities of some present-day computers are so beautiful and so powerful that it seems unappreciative to speak of limiitations. On the other hand, present-day graphics are simply unable to cope, at any rate in a cost-effective way, with the levels of detail or amounts of information in large, complex programs, let alone processes. [Consider the (stationary, static) sequence of instructions that is still sometimes written on paper to be the program, the dynamic sequence of actions that develops inside the computer when the program "runs" to be the process.] On balance, it seems best to risk seeming unappreciative and to think for a short time about the limitations.

Another approach to the limitations of graphics is historical. In 1959, one of us had the early privelege of working at a computer console with a display capable of resloving a million distinct points. The console on which this is being written, which is "state of the art", also has a million pixel display. There is the difference that the display of 26 or 27 years ago was considerably more expensive. Also, it was a vector display, whereas the present one is a raster-scan display. (It is not clear which is better for diagramatic graphics.) In any event, when you compare the capabilities of the computers as computers, the comparison favors the present day overwhelmingly -- by a factor of a few thousand in effectiveness per unit cost -- but when you compare their graphic display capabilities, you have to conclude that things have gotten better graphically, over the 26- or 27-year period, by not more than a factor of 10 or 20.

1. Applications of graphics to programming require more and more graphical capability as the target programs get larger and more complex. To display large programs graphically will require much more powerful graphic displays. According to our present picture of what is required, a minimal graphics system

for an experienced explicit programmer should have about 40 million pixels of display area, which might be divided into 1 to 10 display surfaces of 40 million to 4 million pixels each. That is a very stiff requirement by today's standards, calling for ten times the display investment of an up-scale workstation or for hardware that does not exist. As noted, the graphic display situation has not been improving rapidly, at least as measured in terms of pixel count, but a single 40-million-pixel display could probably be developed in a very few years, if there were a concerted campaign to do so, and the kind of workstation we are envisioning would then be economic. Perhaps, indeed, a concerted campaign is under way: it appears that rapid strides are being made in Japan in the development of larger and clearer flat displays, such as liquid crystal and electroluminescent displays, and that applications in television may further stimulate their development.

2. It may be a limitation of present-day graphics that displays are mounted vertically, and it is surely a limitation that there is not, in widespread use, a light pen with which the user can sketch and write as well as with a ball-point pen. The two things are no doubt linked: if there were good, inexpensive displays oriented in such a way that people could draw and write on them conveniently, then there would surely be good, inexpensive light pens, and possibly *vice versa*. A large desk-surface display and a high-resolution light pen are key parts of every graphical programming or monitoring workstation that we have envisioned.

3. A workstation for a graphical programming system or graphical monitoring system for an inexperienced programmer or user should be, in our judgment, as powerful as a present-day LISP machine or perhaps a VAXstation II. It may take only one more step of cost reduction to make that much power quite economically available. The Commodore Amiga and the IBM PC-AT and its work-alikes seem to be right on the threshold. In principle, they can be equipped with enough memory, and they are fast enough. What they need most, we believe, is more display resolution. Perhaps it will be possible to do the job with pop-up or pull-down menus and rapid substitution of one part of the situation for another, but clearly things will be better if a whole "theatre of operations" can be shown on a single display.

4. Applications of graphics to programming require rather large address spaces. Computers such as the existing Apples (512 kilobytes or less of main memory) and the IBM PC and XT (640 or less) are nowhere near adequate in terms of either physical memory or address space. The Amiga and the AT and work-alikes will in principle accomodate 15 or 16 megabytes of semiconductor memory without having to resort to bank-switching, and that, or even half that, is good for the applications we foresee -- short of trying to deal with very large programs and to help highly experienced programmers. The main limitation that exists and restricts use of some of the machines just mentioned is that it is difficult actually to fit, say, 8 megabytes of memory into the space actually available right now -- or that the add-on arrangements are not quite yet actually available.

5. It is not clear that we should list the speed of present-day processors as a limiting factor. In the development of GRAPPLE, even on the VAX-750 with only one user, speed was always a problem, but it has been possible to make it a

tolerable problem by paying a little more attention to software efficiency, and it may be that much better efficiency-oriented programming would solve the problem entirely. It appears that GRAPPLE runs somewhat faster on the VAXstation II than on the VAX-750. In any event, it is clear that computers are rapidly getting faster. Our feeling is that the order of criticality is: (most) display capacity, (then) size of physical memory, and (least) processor speed.

8. Conclusions

On the basis of our experience with graphical programming and monitoring in this project, the main conclusion is that, given the graphical capabilities of computers that will be affordable, during the next decade, for programming and monitoring the interpretation of programs, the most promising lines of development are those aimed at helping inexperienced programmers and users of programs and, especially, people (such as users of spreadsheets, data bases, and modeling programs) who may be thought of as *implicit* programmers. This conclusion is expressed more fully, along with conclusions about other issues, in the subsection on Conclusions About Dynamic Graphical Representation of the Evaluation of Programs.

Graphical representation has several negative features, such as taking up more display and memory space and more processing cycles than symbolic representation, but graphical representation does make programs and data seem more concrete and definite that does symbolic representation.

With respect to the particular approach represented by GRAPPLE, it would probably be wrong for the writer to draw a firm conclusion because the writer, having spend literally hundereds of hours with GRAPPLE and doubtless suffering from some pride of authorship, is not in a good position to evaluate it. What GRAPPLE needs, now, is evaluation as a concept demonstration by a few people who know something about LISP programming but are not so advanced as to have no need at all for concrete imagery.

9. Recommendations

The Department of Defense should continue to explore graphical programming and monitoring, especially as aids to personnel learning to use or program computers.

The Department of Defense should not be very hopeful in the near term about uses of dynamic graphics, of the kind we have explored, to amplify the effectiveness experienced and skilled programmers. However, another factor of ten in computer capability might cancel out that pessimistic suggestion.

A running GRAPPLE should be made available to program managers of the DARPA Informtion Processing Techniques Office, at DARPA in Arlington, Virginia, with the hope that some of them, and perhaps visitors to IPTO, will have time to make informal assessments of GRAPPLE. (Arrangements toward that end are being made with Dr. Stephen Squires of IPTO.) The assessors should keep in mind, of course, that the present GRAPPLE is a concept demonstration program and not a robust or polished product.

10. References

Dike, Jeffrey G., *An Interface Between MIM and the X Window System*, **Undergraduate Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1986.**

Galley, S. W., and Greg Pfister, *The MDL Programming Language*, **Laboratory for Computer Science (No report number), Massachusetts Institute of Technology, Cambridge, Mass., 1979.**

Naylor, Catherine, *Graphical Representation of MDL Programs as Trees*, **Undergraduate Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1984.**

Po, Lim, *DIGRAM, Device Independent Graphics for MDL*, **Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1982.**

Shrivanandon, John, *The Representation of Programmer-defined Data Types in Graphical Programming*, **Undergraduate Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1985.**

Thompson, Michael A., *Graphically Representing the Control Structure of MDL Programs*, **Undergraduate Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1983.**

11. List of Abbreviations, Acronyms, and Symbols

Here is an alphabetized list of the abbreviations, acronyms, and symbols used in the report. Associated with each acronym, abbreviation, and symbol is an expansion or explanation.

+++	Put additional operators into the operator menu or additional data types and classes into the menu of data types and classes.
0?	A MDL operator. Is the argument equal to 0?
1?	A MDL operator. Is the argument equal to 1?
=	GRAPPLE term for the MDL operator SET.
==	GRAPPLE term for the MDL operator SETG.
==?	A MDL operator. Are the two arguments identical (i.e., the same object?)
=?	A MDL operator. Are the two arguments equal?
?DtG	Expand an abbreviation or explain an item in the menu of data types and classes.
?Main	Expand an abbreviation or explain an item in the main menu.
?R	Expand an abbreviation or explain an item in the operator menu.
A	GRAPPLE term for the MDL operator FORM. Abbreviation for *Applier*.
ASC	GRAPPLE abbreviation for the MDL operator ASCII.
AT	IBM PC-AT. Advanced Technology. Personal Computer.
ATN	GRAPPLE abbreviation for the MDL operator ATAN.
BBN	Bolt Beranek and Newman, Cambridge, MA.
BNL	GRAPPLE term for the MDL operator CRLF. Begin New Line.
C	GRAPPLE term for the MDL operator STRING. Character string. Also a computer language.
CMP	GRAPPLE term for the MDL operator STRCOMP. CoMPare.
CND	GRAPPLE term for the MDL operator COND. CoNDitional.
COND	A MDL operator. CONDitional.
COS	A MDL operator. COSine.

C^ — GRAPPLE term for the MDL operator PNAME. Character-string name.

Chkpt — An item in the main menu. Lets the programmer record the state of GRAPPLE at a particular point in time (colloquially, *create a checkpoint*) or go back to a state *checkpointed* earlier.

DARPA — Defense Advanced Research Projects Agency.

DECL — A MDL data type. DECLaration.

DFR — An item in the menu of operators, used in order to defer the definition of a subordinate operator the name of which is being used in a definition. Forward reference. DeFeR.

DIGRAM — A graphics system for MDL. Device Independent GRAphics for MDL.

Decl — An item in the main menu. Insert declarations into the argument list of the function or macro most recently created.

Down — An item in the main menu. Make a miniature copy of the most recently prepared function or macro and put it down on the shelf.

DtC — Data types and Classes. A menu in GRAPPLE.

EM? — GRAPPLE abbreviation for the MDL operator EMPTY?

ESC — A character in the American Standard Code for Information Interchange. ESCape.

Embed — An item in the main menu. Select a location in the object that is in the central programming square. Select a miniature structure from the shelf. Select a top-level expression from the body of the function or macro represented by the miniature structure. Embed that expression in the object in the central programming square at the point selected.

Erase — An item in the main menu. Erase the central programming area, the main menu, and the residence sites for global variables, function and macro parameters, and local variables.

ErrRt — An item in the main menu. Error return. Return to the top level of the MDL interpreter. (For use after an error condition has arisen.)

Eval — An item in the main menu. Evaluate the expression most recently programmed. This evaluation uses as the values of parameters the values provided as *illustrative values* or *default values* or *local values* as the expression was being programmed. `Eval` ceases to be available for evaluation of an expression at the time the programmer declares that he has finished it by selecting `FinIF` or `FinIX`.

EXP — A MDL operator. EXPonentiate.

FONTED — A Font Editor used in creating icons to represent programming constructs.

FiniF — An item in the main menu. Finish function. The programmer has completed the body of a function or macro, so package the body and the argument list into a function or a macro and get a name for the new function from the programmer.

FiniX — An item in the main menu. Finish expression. The programmer has completed a top-level expression, so rearrange the display in the central programming area in preparation for the programming of another top-level expression.

G — Abbreviation for GRAPPLE. The name of the MDL function used to start up GRAPPLE.

G=? — A MDL operator. Greater than or equal to?

G? — A MDL operator. Greater than?

GBIND — A MDL data type.

GRAPPLE — GRAPhical Programming LanguagE.

Glb — An item in the main menu. Create a global datum with a residence site in the global data area.

Glbs — An item in the main menu. If the global data have been removed from the screen, return them to their residence sites and redisplay them.

I-C — GRAPPLE term for the MDL operator ISTRING.

I-L — GRAPPLE term for the MDL operator ILIST.

I-U — GRAPPLE term for the MDL operator IUVECTOR.

I-V — GRAPPLE term for the MDL operator IVECTOR.

IBM — International Business Machines Corporation.

Init — An item in the main menu. Present a menu for initialization that is like GRAPPLE's initial menu except that it works with the mouse and includes, in addition to the five alternatives of the initial menu, three special reinitialization alternatives. They are: (1) to reinitialize the mechanism for deferred definition, i.e., for forward reference to a function or macro not yet defined; (2) to reinitialize the situation as regards global data; and (3) to reinitialize the arrangement that keeps track of the component diagrams have been used and prevents confusion between what has been used and what GRAPPLE will create to represent new programming strucutres. The third special reinitialization is drastic: in

	effect, it destroys all the graphic structures that have been defined thus far.
L	GRAPPLE abbreviation for the MDL operator LIST.
L=?	A MDL operator. Less than or equal to?
L?	A MDL operator. Less than?
LBIND	A MDL data type.
LISP	A family of programming languages. LISt Processing language.
LN	GRAPPLE term for the MDL operator LOG. Logarithm Natural. LOGarithm.
LNG	GRAPPLE term for the MDL operator LENGTH.
M.I.T.	Massachusetts Institute of Technology, Cambridge, MA.
MAX	A MDL operator. MAXimum.
MDL	A programming language. An outlying member of the LISP family of programming languages. More Datatypes than Lisp.
MEM	GRAPPLE abbreviation for the MDL operator MEMBER.
MIMC	A *MDL compiler. Machine Independent Mdl Compiler.*
MIN	A MDL operator. MINimum.
MOD	A MDL operator. MODulo.
MP1	GRAPPLE abbreviation for the MDL operator MAPF. MAP First.
MPL	GRAPPLE abbreviation for the MDL operator MAPLEAVE.
MPR	GRAPPLE abbreviation for the MDL operator MAPR. MAP Rest.
MPS	GRAPPLE abbreviation for the MDL operator MAPSTOP.
MP_	GRAPPLE term for the MDL operator MAPRET. MAP RETurn.
MSUBR	A MDL datatype. Mediated SUBRoutine.
Macro	An item in the main menu. Give the programmer a chance to make the next-constructed top-level object be a macro instead of a function (or to change his or her mind and go back to the default, which is to create a function).

MuGrP — An item in the main menu. To the menu of operators from graphical programming. Put the name of the function or macro most recently defined by graphical programming into the menu of operators (the R menu) and create a data-base entry for it.

MuMDL — An item in the main menu. To the menu of operators from MDL. Put the name of a specified MDL object into the menu of operators and create a data-base entry for it.

New-t — An item in the main menu. Prepare to define a new data type.

O — GRAPPLE term for any MDL object.

OBLIST — A MDL data type.

PAR — GRAPPLE abbreviation for the MDL operator PARSE.

PC — Personal Computer.

PC-AT — IBM Personal Computer-Advanced Technology.

PR — GRAPPLE abbreviation for the MDL operator PRINC. PRINt Character.

PRG — GRAPPLE abbreviation for the MDL operator PROG. PROGram.

PRI — GRAPPLE abbreviation for the MDL operator PRIN1. PRINt 1 object.

PROG — A MDL operator. PROGram.

PRT — GRAPPLE abbreviation for the MDL operator PRINT.

R — GRAPPLE abbreviation for operator. operatoR. Also, a column in the residence area for parameters and local values. Required parameter.

RD — GRAPPLE abbreviation for the MDL operator READ.

RPT — GRAPPLE abbreviation for the MDL operator REPEAT.

RST — GRAPPLE abbreviation for the MDL operator REST.

RTN — GRAPPLE abbreviation for the MDL operator RETURN.

Ratch — An item in the main menu. Ratchet. Ordinarily, the data base of operators that GRAPPLE knows about is initialized, *via* initialization 0 or initialization 1, by being copied from a data base called the *initial data base*. Selecting the item Ratchet replaces the initial data base with a copy of the currently working data base, which contains not only the information from the initial data base but also information about

functions and macros the programmer has constructed. Ratchet thus updates the inital data base to include the newly programmed objects. The only way to get back to the "initial initial" data base after ratcheting is to load a file. The file can be either an old *save* file that contains the "initial initial" data base (which returns the whole graphical programming system to the state recorded by the *saving*) or a text file that contains a copy of the "initial initial" data base. The file `initial-grapple-data-base.mud` contains a copy.

ReDrw — An item in the main menu. ReDraw the program diagram corresponding to the current program tree. This assumes that a hiatus has been introduced by the use of the `Erase` menu item.

Res-t — An item in the main menu. Result type. Accept the designation of a data type or class and then ensure that the next function or macro constructed by graphical programming will return an object of that type or class.

Retrv — An item in the main menu. Retrieve. Retrieve the contents of a Unix file, the name of which is to be specified. In MDL terms, the contents will be `FLOAD`ed into the MDL interpreter. If the retrieved file is one created by the `File` menu item, then it should be a file of information from the present programming series. Otherwise, some of the structures in the file may overwrite structures in the present system.

SIN — A MDL operator. SINe.

sq — Square.

SQT — GRAPPLE abbreviation for the MDL operator SQRT.

Save — An item in the main menu. Save -- i.e., checkpoint, i.e., record the current state of the system -- in a MDL *save* file with the filename that the user will specify (and with the extension `.save`). This operation takes about a minute.

Sgmnt — An item in the main menu. Give the programmer a chance to make the next object to be constructed by graphical programming be a MDL segment (consisting of the contents of a structured object without the container) instead of (if indeed it is going to have contents) a full-fledged structured object. (Also give the programmer a chance to make the system revert to the default state, which is to keep the containers of structured objects.)

TYI — A MDL operator. TYpe In a character.

U — GRAPPLE abbreviation for the MDL operator and data type UVECTOR.

UNP — GRAPPLE abbreviation for the MDL operator UNPARSE.

UVECTOR	A MDL operator and data type.
Ulrix	A Digital Equipment Corporation operating system for VAX computers. A version of Unix.
Unix	An operating system developed at the Bell Telephone Laboratory and widely used on VAX (and many other) computers.
V	GRAPPLE abbreviation for the MDL operator and data type VECTOR.
VAL	GRAPPLE term for the MDL operator and data type GVAL.
VAX	A line of computers manufactured by Digital Equipment Corporation.
X	GRAPPLE term for any structured object.
X?	GRAPPLE term for the MDL operator STRUCTURED?
XIT	An item in the operator menu and in the menu of data types and classes. eXIT.
XT	An IBM personal computer, and eXTension of the PC.
f	GRAPPLE abbreviation for the MDL operator and data type FLOAT.
i	GRAPPLE term for the MDL operator and data type FIX. integer.
iTH	GRAPPLE term for the MDL operator NTH.
m?	GRAPPLE abbreviation for the MDL operator MONAD?
mem	GRAPPLE abbreviation for the MDL operator MEMQ.
mudsub	A command to Unix to load a MDL *save* file.
or	GRAPPLE term for the MDL operator OR.
pict	A GRAPPLE diagram. picture.
q	GRAPPLE abbreviation for the MDL operator QUOTE.
sos	sum of squares.
t	GRAPPLE abbreviation for the MDL operator TYPE.
t?	GRAPPLE abbreviation for the MDL operator TYPE?
tCChk	An item in the main menu. Type and Class Checking. Turn *on* or *off* the mechanism that GRAPPLE uses to make sure that the data types

and classes specified in the program under construction are compatible with the operators that are specified. Even when this checking mechanism is turned off, the basic type-checking mechanism of MDL will be operating -- unless it, too, has been turned *off*. To turn it *off*, type <DECL-CHECK %<>> and press the DO-IT key. To turn it *on*, type <DECL-CHECK T> and press the DO-IT key.

val GRAPPLE term for the MDL operator and data type LVAL.

s GRAPPLE term for the MDL operator FLOAT and data type FLOAT.

DISTRIBUTION LIST

addresses	number of copies
Richard M. Evans RADC/COES	20
RADC/DOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	12
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
HQ AFSC/DLAE ANDREWS AFB DC 20334-5000	1
HQ AFSC/XRK ANDREWS AFB MD 20334-500	1
HQ AFCTEC (OAWC) Attn: Capt. Novack) KIRTLAND AFB NM 87117-7001	1
ASD-AFALC/AXP WRIGHT-PATTERSON AFB OH 45433	1

ASD/AFALC/AXAE 1
Attn: W. H. Dungey
Wriight-Patterson AFE OH 45433-6533

AFIT/LDEE 1
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

AFWAL/MLPO 1
WRIGHT-PATTERSON AFB OH 45433-6533

COMMAND CONTROL AND COMMUNICATIONS DIV 2
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CODE DICA
QUANTICO VA 22134-5080

AFLMC/LGY 1
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114

COMMANDING OFFICER 1
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46219-2189

COMMANDING OFFICER 1
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 9642B
SAN DIEGO CA 92152-5000

COMMANDER (CODE 3433) 1
ATTN: TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001

SUPERINTENDENT (CODE 1424) 1
NAVAL POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER 2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

ESD/XRS 1
ATTN: ADV SYS DEV
HANSCOM AFB MA 01731-5000

ESD/ICP 1
HANSCOM AFB MA 01731-5000

ESD/XRSE 2
BLDG 1704
HANSCOM AFB MA 01731-5000

HQ ESD SYS-2 1
HANSCOM AFB MA 01731-5000

The Software Engineering Institute 1
Attn: Major Dan Burton, USAF
580 South Aiken Avenue
Pittsourgh PA 15232-1502

DoD COMPUTER SECURITY CENTER 1
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

J. C. R. Licklider 5
Massachusetts Institute of Technology
Laboratory for Computr Science
77 Massachusetts Avenue
Cambridge, MA 02139

ESD-MITRE Software Center Library 2
c/o Ms J. A. Clapp
MITRE Corp D-70, MS A-359
Burlington Road
Bedford MA 01730

Software Engineering Institute Technical Librar 2
Carnegie-Mellor University
Pittsburgh PA 15232
ATTN: Korola Fuchs

Col J. Green 1
Dir, STARS JPO
Room C-107
1211 South Fern Street
Arlington VA 22202

☆ U.S. Government Printing Office: 1988 511-117/64086

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.